Side-Channel Attacks on Open vSwitch

Daewoo Kim
University of Waterloo

Sihang Liu
University of Waterloo

Abstract

Virtualization is widely adopted in cloud systems to manage resource sharing among users. A virtualized environment usually deploys a virtual switch within the host system to enable virtual machines to communicate with each other and with the physical network. The Open vSwitch (OVS) is one of the most popular software-based virtual switches. It maintains a cache hierarchy to accelerate packet forwarding from the host to virtual machines. We characterize the caching system inside OVS from a security perspective and identify three attack primitives. Based on the attack primitives, we present three remote attacks via OVS, breaking the isolation in virtualized environments. First, we identify remote covert channels using different caches. Second, we present a novel header recovery attack that leaks a remote user's packet header fields, breaking the confidentiality guarantees from the system. Third, we demonstrate a remote packet rate monitoring attack that recovers the packet rate of a remote victim. To defend against these attacks, we also discuss and evaluate mitigation solutions.

1 Introduction

Side-channel attack is a category of attacks that exploits extra information from unintended channels rather than vulnerabilities in software or algorithms. In computer systems, side-channel attacks threaten and intercept valuable information through both hardware and software. Caches are common targets of side-channel attacks, as their hit/miss timing can be used to infer secret information about the program, bypassing the existing system-level isolation and protection. For example, CPU caches [16,17,21,28,64] and TLBs [15] can be used to infer program behavior and even data values executed by the program; database caches [18,49] leak database queries and can be used to construct the database. To perform side-channel attacks on caches, the attacker typically initializes the cache to a known state and then observes any changes in the state made by the target application that shares the same cache. For instance, Prime+Probe [28] and Flush+Reload [64] are common approaches.

As modern systems are being moved to the cloud for better efficiency and maintenance, resource sharing becomes more common. At the same time, such sharing scenarios are more susceptible to side-channel attacks, as different applications and services can be co-located on the same server, sharing not only hardware components but also part of the software stack. Prior works have demonstrated side-channel attacks in remote cloud scenarios. For example, NetSpectre [47] proposes a remote Spectre attack that reads memory over the network using Evict+Reload, and NetCat [24] exploits the Data-Direct I/O (DDIO) to infer the last-level cache hit/miss over the network using Prime+Probe. These remote side-channel attacks are stealthy as an attacker can leak secret data over the network.

In a cloud environment, virtualization is a common approach to allow multiple users to share the same server, where the hypervisor manages a number of virtual machines (VMs). The network that connects these VMs is usually also managed by network virtualization systems that apply network policies to incoming packets and use software switches to forward packets to different VMs by bridging them between the physical network interface and the VMs. The Open vSwitch (OVS) is one of the most widely used software switches [41,54] that follows the OpenFlow standard [33]. Based on the packet flow that travels from a source to a destination, OpenFlow specifies actions that an incoming packet will take (such as being forwarded to a certain port) by looking up a number of OpenFlow tables in sequence. OVS accelerates the sequential table lookup process with two levels of software-based caches — a fast but small *microflow cache* (with a default size of 8192 as of OVS v3.6 with the DPDK datapath) and a slower but larger *megaflow cache*. The caches reside in the main memory, containing structures that match the network packet header and actions that the matching packets will take. *Hit* in either cache is referred to as the *fast path*. An incoming packet first looks up the microflow cache. A microflow miss will be directed to the megaflow cache. Eventually, megaflow cache misses will perform the slow OpenFlow table lookup,

which is referred to as the *slow path*. In summary, OVS is a complex software system with caching, intended to be shared among VMs, and can be accessed remotely via network.

In this work, we aim to examine and assess the security implications of OVS — whether attackers can violate the confidentiality guarantees in the virtualization system and obtain unauthorized access to secret information from remote users through OVS. There have been prior security studies on OVS systems. However, they only focus on performance degradation due to malicious users that can lead to denial-of-service (DoS) [1–4, 9]. However, the side channel aspect largely remains unstudied.

To exploit side channels in OVS, a key prerequisite is a good understanding of its characteristics and behaviors. Although OVS is open-sourced, its security implications remain unclear. Therefore, we characterize OVS from a security angle, by evaluating latencies, eviction and timeout policies, and internal structures of each cache. And then, we cross-reference our findings with the OVS documentation. We summarize our findings as three attack primitives. (1) Each OVS component has distinguishable latencies: microflow cache, megaflow cache, and slow path in OpenFlow feature round-trip times (RTTs) of 270.08 µs (with default microflow cache size), 277.18 µs, and 1196.03 µs, respectively, in a one-hop network system. The latency differences can be exploited to infer cache states. (2) The microflow cache evicts an existing flow entry upon a collision with the hash of a new packet header. As the hash value is generated from five header fields (IP addresses and port numbers of source and destination, and protocol), the microflow cache can leak the packet header fields. (3) The megaflow cache keeps a number of subtables for incoming packets to look up and determine their actions. The lookup is sequential and completes upon the first hit, and frequently accessed subtables are ordered at the front. Thus, the access frequency controls the subtable ordering and affects latency, which can be leveraged to infer packet rate. Next, we present three attacks using these attack primitives.

First, we establish and demonstrate remote covert channels using both the microflow cache and the megaflow cache. Such covert channels involve a remote sender and receiver who do not have direct communication. Instead, the sender has access to a service (Memcached in our experiment) that is co-located with the receiver. The service and the receiver are running in separate VMs but share an OVS. To send secret data, the remote sender queries the service to generate contention on the OVS. In parallel, the receiver queries another service located in a separate server to sense this contention. The microflow-cache-based covert channel leverages the latency difference between microflow cache hit and miss. Depending on the bit that the sender transmits, the sender either generates a collision on a specific microflow entry or not, which affects the receiver's RTT and can be detected. The megaflow-cache-based covert channel exploits the latency difference among subtables and their reordering mechanism in the megaflow cache. The sender controls the packet rates to order the target subtable at the desired location. The location can then be detected by the receiver based on the latency difference, thus retrieving the bit. Covert channels based on the microflow and the megaflow caches achieve 15.8 bit / s and 0.73 bit / s, respectively, with low error.

Second, we demonstrate an attack that retrieves a remote user's IP address and port number via the microflow cache. We also assume that the attacker can query a service that is co-located with the victim and shares the OVS. We assume that the victim is accessing a publicly accessible service that is known by the attacker. Thus, among the five packet header fields that generate a hash for the microflow cache index, the destination IP + port, and the protocol are known. The attacker only needs to recover the victim's source IP + port based on hash collisions. The attacker performs a Prime+Probe attack to identify entries that are evicted by the victim, by measuring the RTTs of all microflow cache entries via queries to the co-located service. Then, the attacker computes the hashes of all possible victim's IP and port combinations to find the one that matches the hash values of the evicted entries. By probing the victim for 7 min (with default microflow cache size), the attacker can successfully recover these fields at an accuracy of 91 %, breaking the isolation among virtual machines.

Third, we present a packet rate monitoring attack, where an attacker infers the packet rate of a remote user. We assume that a similar scenario as the packet header recovery attack, where the attacker can query a publicly accessible service that is co-located with the victim and shares the OVS. The victim sends packets to another remote service that is known to the attacker, e.g., can be detected using the packet header recovery attack. The attacker exploits the packet-rate-dependent reordering of the megaflow cache subtables to recover the victim's packet rate. The attacker first evicts the victim's flow in the microflow cache, which results in the increment of the megaflow subtable hit counts corresponding to the victim's traffic. In parallel, the attacker orders other subtables by accessing them at a known rate, and probes the victim's subtable. The relative latency difference between the other subtables and the victim-accessed subtable reveals the relative packet rate difference. This way, the attacker can track the range of the victim's packet rate in real time. Our evaluation with real-world network traces demonstrates a 71.92 % monitoring success rate (with default microflow cache size).

Finally, we propose and evaluate three mitigation mechanisms for OVS side channels: isolation of OVS instances, hash randomization for the microflow cache, and subtable reordering randomization for the megaflow cache. Our evaluation shows that these methods effectively mitigate the side channels we identified in this work.

The contributions are summarized as follows:

- We thoroughly study and characterize caching in OVS from the security aspect.
- We establish remote covert channels using both microflow

cache and megaflow cache in OVS.

- We present a novel remote packet header recovery attack that retrieves a remote user's IP address and port number from network packets.
- We demonstrate a novel remote packet rate monitoring attack that identifies a remote user's real-time packet rate.
- We propose and evaluate three defense mechanisms for OVS, demonstrating effective mitigation of side channels.

2 Background

In this section, we first introduce side-channel attacks, and then OpenFlow and Open vSwitch (OVS).

2.1 Side-Channel Attacks

Side channels are communication channels based on indirect, unintended behaviors and features. In computer systems, side channels widely exist. Attackers can perform side-channel attacks by leveraging these channels to secretly leak information about the target system. For example, an attacker can exploit power [44, 59, 66] and thermal signals [14, 23] of computing systems to bypass the original protections and isolation. A secret sender may also transmit information using such unintended channels (i.e., covert channels).

One of the major categories of side-channel attacks exploits timing differences of caches in computer systems. Caching is a common technique to provide fast access to commonly used data, such as caches in CPUs and databases. Accessing data in a cache (i.e., cache hit) has lower latency than those not in the cache (i.e., cache miss). An attacker can infer secret information about programs using such timing differences. There are several common methods to perform side-channel attacks on caches. For example, Prime+Probe [28] leaks information about whether a victim uses a certain cache set. First, the attacker fills a cache set with its data and then measures the time to read the data. If the victim has accessed the same cache set, the attacker's data is evicted, causing longer reaccess time. Flush+Reload [64] exploits shared memory (e.g., a shared library) and leaks information about whether a victim program loads specific data into the cache. The attacker flushes a targeted memory location and measures the reload time. If the victim has loaded the corresponding cache line into the cache, the reload time becomes shorter.

Even more stealthily, cache-based side-channel attacks utilizing these techniques can be executed over a network. Examples include NetSpectre [47] and NetCAT [24], which demonstrated that the caching timing differences are significant enough to be discerned even after traversing a network. As cloud systems commonly share computing platforms over the network, the capability of launching side-channel attacks over the network is a prominent threat.

Although side channels widely exist and are hard to detect, performing a side-channel attack has a key prerequisite, that

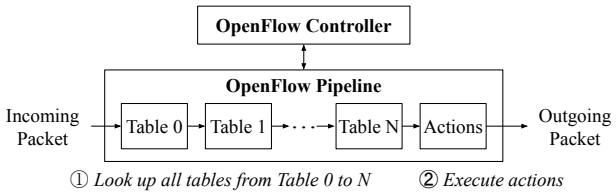


Figure 1: Packet forwarding in OpenFlow pipeline.

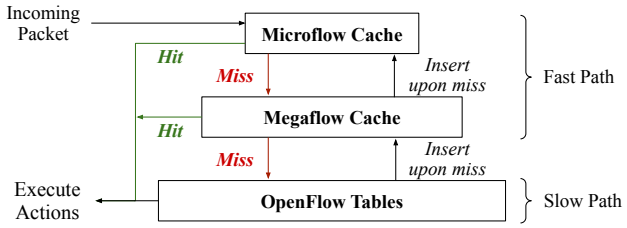


Figure 2: Software-based caches in OVS.

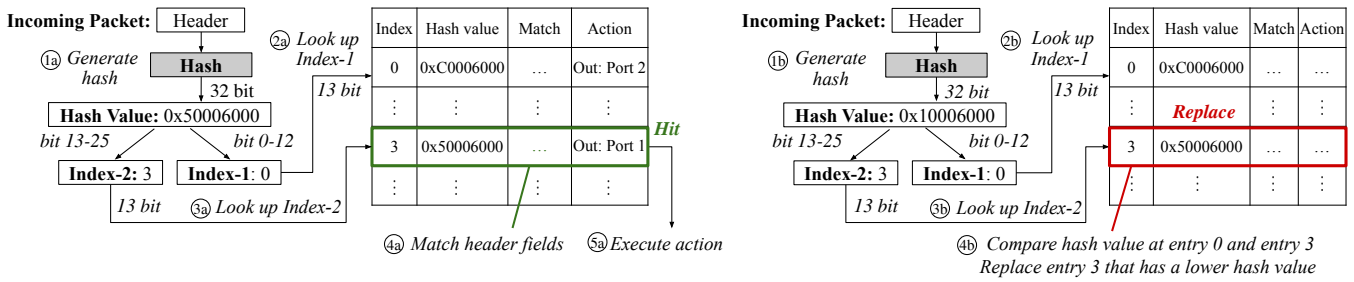
is good knowledge about the target system. For example, performing a side-channel attack on the CPU cache requires reverse-engineering of CPU cache structure (e.g., size and associativity) and its latency [16, 17, 21, 28, 64]; one that targets a database system would need full knowledge about the database algorithm and software implementation [18, 49].

2.2 OpenFlow and Open vSwitch (OVS)

Conventionally, network devices have fixed functions to control network traffic. Therefore, each function in a device has to be separately maintained to control network traffic. In this section, we introduce OpenFlow and Open vSwitch which alleviate the burden of network management.

2.2.1 OpenFlow

OpenFlow [33] is a software-defined networking (SDN) control protocol. Figure 1 demonstrates how a switch following OpenFlow protocol works. The OpenFlow controller is a centralized unit that decides the routes of packet flows to the final destinations. A packet flows from a source to a destination via the route decided by the controller in OpenFlow protocol. Flow tables decide the *action* that incoming packets will take by *matching* packet header fields with each table. If the requested flow information does not exist, the OpenFlow controller updates the flow tables based on the user-specified OpenFlow rules. Figure 1 demonstrates the OpenFlow pipeline, where an incoming packet goes through tables (Table 0 to N) in order (step ①). Eventually, the packet takes actions based on the table lookup at the end of the pipeline (step ②), such as forwarding the packet to a certain port.



(a) Lookup an existing flow.

(b) Insert a new flow and replace an existing flow.

Figure 3: Microflow cache (with default size of 8k entries).

2.2.2 Open vSwitch

The Open vSwitch (OVS) [41, 54] is one of the most widely-used virtual switches based on the OpenFlow protocol. Like physical switches that follow the OpenFlow protocol and go through table lookups, OVS does the same for virtualization environments, being managed by the OpenFlow controller. In virtual environments, OVS establishes connections among virtual machines (VMs) and the physical NIC on the host.

The conventional OpenFlow table lookups always happen sequentially for all incoming packets, slowing down packet forwarding. To accelerate the sequential OpenFlow table lookup process, OVS maintains two levels of caches. These caches are implemented in the software and kept in the main memory. Figure 2 shows the cache hierarchy inside OVS, where the *microflow cache* is the first structure to be accessed upon an incoming packet, followed by the *megaflow cache* if the microflow cache misses. Accesses within the caches are referred to as the *fast path*. Only when both caches miss, the packet is directed to the OpenFlow tables, which is the *slow path*. Much like caching in CPUs, once the packet is handled by a lower-level cache in the hierarchy, its corresponding flow will be inserted into the upper levels of caches.

2.3 Caching in Open vSwitch

OVS contains a small but fast *microflow cache* for exact matching and a larger but slower *megaflow cache* that uses wildcard rules for matching, organized into a number of subtables. Next, we describe both caches in detail.

2.3.1 Microflow Cache

The microflow cache is the first level of cache in the hierarchy of OVS, as depicted in Figure 2. The microflow cache has a fixed-size table with 8192 (i.e., 8k) entries by default (as of OVS v2.17.9). Each entry contains one flow. The incoming packet needs to be an exact match with the entry for a cache hit, i.e., all packet header fields need to be matched, according to the "match" field. If an incoming packet hits the microflow cache, it will then take the "action" specified in the entry. By performing only a single, exact hash lookup, the microflow cache is optimized for low latency but with limited capacity.

Figure 3a demonstrates the lookup process in the microflow cache when it has the default size of 8k entries. The microflow cache first generates a 32-bit hash value from five fields (source IP and port, destination IP and port, and protocol) in the packet header (step ①a). The 32-bit hash value is then divided into two hash values, forming two 13-bit hash values for two indices in the default 8k-entry microflow cache: Index-1 using bits 0–12 and Index-2 using bits 13–25 (the remaining higher-order bits are unused). It first looks up Index-1 which corresponds to entry 0 and compares the full 32-bit hash with the entry (step ②a). In this example, entry 0 does not match. Then, the microflow cache looks up Index-2 which corresponds to entry 3 (step ③a) and finds a matching entry (step ④a). Thus, the packet is forwarded to output port 1 according to the action field (step ⑤a). In case of no matching entry in the microflow cache (i.e., a miss), OVS forwards the packet to the next-level megaflow cache. Once a miss is resolved, OVS inserts this flow into the microflow cache. With this mechanism, increasing the number of bits in each hash index enlarges the microflow cache. For example, a microflow cache with 16 384 (i.e., 16k) entries has 14 bits for each index.

According to OVS documentation [41, 54], a flow stays in the microflow cache until it is evicted by a new flow. Figure 3b illustrates the eviction process when inserting a new flow. Similar to the microflow cache lookup, the hash function generates a 32-bit hash value using the packet header (step ①b). Then, it tries to insert the new entry into Index-1 if the entry is available; otherwise, it attempts Index-2. If neither entry is available, the microflow cache compares the hash values of both entries and replaces the one with a smaller 32-bit hash value. In the example of Figure 3b, Index-1 (step ②b) and Index-2 (step ③b) which correspond to entry 0 and entry 3 do not have space available. Therefore, the microflow cache compares the hash values of the two entries (step ④b), and replaces entry 3, which has a smaller hash value than entry 0.

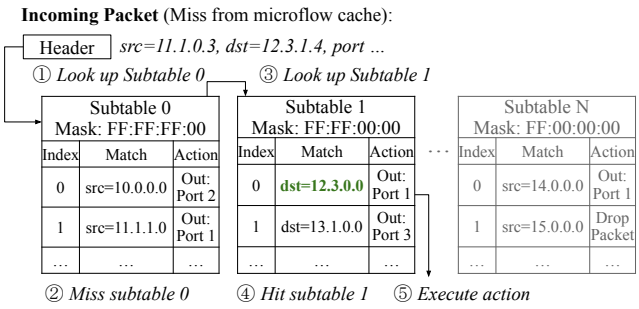


Figure 4: Megaflow cache.

2.3.2 Megaflow Cache

The microflow cache may suffer from a high miss rate in case of massive short-lived connections due to its small capacity. Therefore, OVS integrates a second-level cache, the megaflow cache. It incorporates subtables that represent common combinations of OpenFlow tables, effectively increasing the caching capacity. It inserts a new flow into one of its subtables (or creates a new subtable if it does not exist) upon a megaflow miss. The subtables and entries are created based on the OpenFlow rules. Each subtable in the megaflow cache has unique masks applied to all header fields, and entries in a subtable share the same mask. The megaflow cache looks up subtables in order and executes the corresponding action. However, unlike the OpenFlow table in the slow path that walks through all tables, the megaflow cache stops lookup as soon as it finds the matching entry. There is only one matching entry among all subtables because each megaflow cache entry is unique.

Figure 4 shows an example, where the incoming packet misses the microflow cache, and is thus passed to the megaflow cache. The packet has source IP 11.1.0.3 and destination IP 12.3.1.4. It first looks up subtable 0 (step ①) but misses (step ②). Then, it moves on to subtable 1 (step ③) and hits after applying the bitmask (step ④). Therefore, it skips the remaining subtables (shown in gray in Figure 4) and executes the action from subtable 1 (step ⑤).

The size of the megaflow cache is much larger, containing a number of subtables, unlike the fixed-size microflow cache. To reduce the lookup latency, the megaflow cache reorders subtables by periodically sorting them based on their hit counts. Thus, frequently accessed subtables will be looked up earlier to reduce latency. As the megaflow cache can grow over time, it periodically evicts entries without recent hits.

2.4 Existing Security Studies on OVS

As OVS is intended for sharing among server tenants and attached to the network, security vulnerabilities in OVS may lead to serious consequences. There have been existing security studies on OVS. Csikor et al. [9] perform a tuple space exploration attack on OVS that degrades OVS performance, leading to denial-of-service (DoS) for other OVS users; Šraier investigates potential performance issues in OVS under untrusted network traffic [55]. Additionally, several potential DoS vulnerabilities in OVS have been documented in the CVE system [1–4]. However, these studies only focus on serious performance degradation caused by attackers.

The question we aim to answer is whether an attacker can break the isolation from virtualization, and acquire and transmit secret information via OVS. Prior studies have shown that shared caches may lead to side channels. As OVS contains a cache hierarchy, much like caches for databases and CPUs, in this work, we assess OVS from a side-channel security angle.

Table 1: System configuration for OVS characterization.

CPU	24 cores, Intel Xeon Cascade Lake
Memory	96 GB
NIC	Intel I210, 1 Gbps
Switch	NETGEAR GS308, 1 Gbps
OS	Ubuntu 22.04, Linux kernel v6.5.0
OVS	OVS (v2.17.9) with DPDK datapath (v21.11.9)

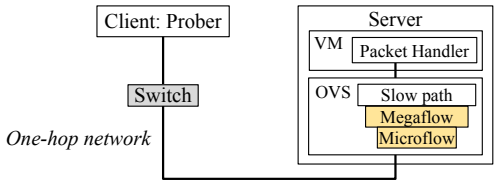


Figure 5: Overall system setup for OVS characterization.

3 OVS Characterization

In this section, we first characterize OVS from a security angle and summarize three attack primitives in OVS.

3.1 Experiment System

Test Platform. The testbed consists of two machines, one client and one server, connected by a one-hop network via a switch. Table 1 lists the system configuration that consists of off-the-shelf platforms and software components. Figure 5 illustrates the experiment setup. The server hosts a virtual machine (VM) that runs a packet handler. A virtual NIC in the VM is connected to OVS and then connects to the physical NIC on the host server. The prober for characterization is located on the client. We use a Memcached instance as the handler to process `GET` requests from the prober. This testbed is isolated to avoid interference with other users. We follow a proof-of-concept setup to demonstrate new side-channel vulnerabilities similar to prior remote [24, 30, 47, 48] and local [15, 17, 28, 37, 53] side-channel attacks.

OVS Configuration. The OVS system uses the DPDK

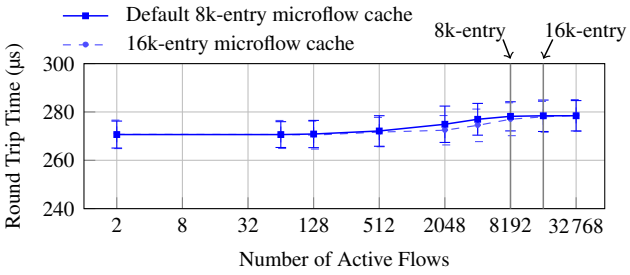


Figure 6: Network RTT of OVS vs. number of active flows.

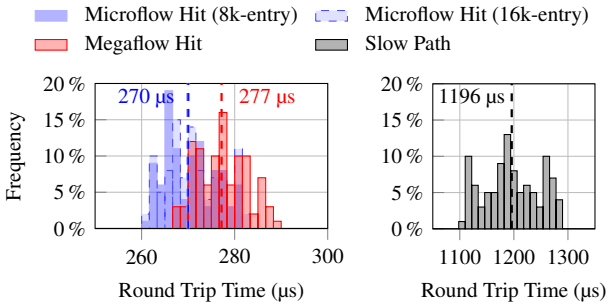


Figure 7: Latency distributions of microflow (8k- and 16k-entry have overlapped distribution), megaflow, and slow path.

library as the data path [38]. Its cache design has not changed since v2.6, with the microflow and megaflow caches in the current version (v3.6, December 2025) still exhibiting the same behavior discussed in Section 2.2. Another OVS version based on the kernel datapath does not include a microflow cache but still has a megaflow cache, which implications will be discussed in Section 3.5. We evaluate both the default microflow cache with 8k entries and a version with 16k entries by scaling the number of bits in the microflow index. We deploy OVS rules from ClassBench-ng [32], a packet classification benchmark commonly used in prior work [34, 52, 56, 57]. These rules form 40 subtables in the megaflow cache. We follow the same configuration in the rest of the paper unless specified.

3.2 Timing of OVS

To understand the network timing of OVS, we first measure the network round trip time (RTT) of OVS by varying the number of active flows (100 times for each). Each round-trip measurement involves a *forward* packet from the prober client and a *backward* response packet, doubling the usage of flow entries in OVS caches. Figure 6 shows the result, where the number of active flows (x-axis) counts both forward and backward directions, and each point is an average RTT (y-axis) with an error bar, representing one standard error ($\pm\sigma$).[1]

We first evaluate the default 8k-entry microflow cache size. We observe that when the number of active flows is low, the latency is the lowest (avg = 270.65 µs, $\sigma = 2.0\,\%$). The latency starts to increase when the number of flows reaches the size of the microflow cache (default 8k), where the average latency is 277.18 µs ($\sigma = 1.9\,\%$). With more flows, the latency is about the same, which is the megaflow cache hit latency. We also evaluate a larger microflow cache with 16k entries. As Figure 6 shows, the microflow cache hit/miss latencies remain almost the same as the default 8k-entry setup, where the hit latency is 270.74 µs ($\sigma = 2.1\,\%$).

Figure 6 does not include the slow path latency because the megaflow cache does not have a size limit. Instead, we measure the first occurrence of a flow as the slow path latency, which has an average latency of 1196.03 µs ($\sigma = 4.3\,\%$). Figure 7 (left) summarizes the RTT distributions of microflow cache hits and megaflow cache hits, showing partially overlapped distributions but still distinguishable with multiple measurements as the average values differ. Figure 7 (right) shows the RTT distribution of the slow path, which is distinct from the microflow and megaflow hits.

Conclusion. The microflow cache, megaflow cache, and the slow path have distinguishable latencies. Repeated measurements enable more reliable differentiation of these latency levels. Specifically, the microflow cache does exact-match, which indicates that observing a microflow cache hit latency guarantees that no other flows use the same cache entry; a megaflow cache hit latency (a microflow miss) indicates that there has been a collision. In comparison, the megaflow cache maintains flows with the same header mask in the same subtable. Therefore, a megaflow hit latency indicates that the subtable is shared with other flows.

3.3 Microflow Cache

We characterize the microflow cache by evaluating the default 8k-entry setup. Because its size does not change its logic or policy, conclusions drawn from these experiments remain the same when scaling the microflow cache size. Then, we summarize the attack primitives based on our findings.

3.3.1 Microflow Cache Timeout

As discussed in Section 2.3.1, the microflow cache only does eviction upon collision. We confirm whether the microflow cache ent ries time out in this experiment. First, the client creates an entry (i.e., a flow) in the microflow cache by sending an arbitrary packet. Then, we measure the RTT of the same flow after different waiting times by resending the packet. We repeat this experiment 100 times under different waiting time intervals. As Figure 8 shows, the RTT remains almost the same (avg = 270 µs, $\sigma = 1.4\,\%$) even after 100 s. Therefore, the microflow cache entries do not time out.

[1] Error bars in the remainder of this paper follow the same format.

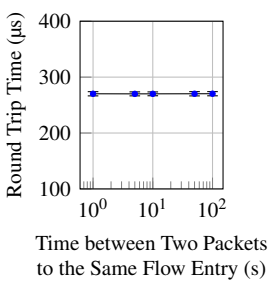


Figure 8: Microflow cache hit latency vs. waiting time ($n = 100$ per data point, 8k-entry microflow cache).

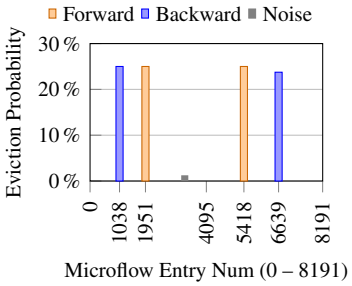


Figure 9: Microflow cache eviction (default 8k entries). Eviction packet is first sent to server (*forward*) and then triggers response (*backward*).

3.3.2 Collisions in Microflow Cache

The microflow cache evicts by collision upon new flows, as introduced in Section 2.3.1. An evicted flow will be a miss if accessed again, leading to longer latency (as shown in Figure 7). We investigate its behavior using the setup in Section 3.1. The prober periodically measures the latencies of all 8k microflow cache entries on the host. Also, the prober sends an additional eviction packet to the packet handler on the server in between every 8k RTT measurements to generate microflow cache evictions. Like the experiment in Section 3.2, the additional packet also leads to two flows: forward and backward. As discussed in Section 2.3.1, each flow triggered by a network packet looks up two microflow cache entries and replaces one of them according to their hash values. Therefore, to demonstrate evictions of both hash indices, we control the hash value of the probing packets by setting the IP address and port number. First, the prober generates packets to probe all microflow entries with the same header fields for 100 times for higher accuracy and chooses 2 entries with the highest average RTT. Then, for the 2 chosen entries that collide with the eviction packets, the prober modifies their IP and port number to get higher hash values than any existing hash values in the microflow cache. This way, these entries will not be evicted, but instead have their other correspondent indices evicted. Again, the prober performs probing with new hash values for another 100 times and chooses 2 entries with the highest average RTT. We evaluate this 20 times and summarize the probability of evictions among 8192 microflow cache entries in Figure 9. Because the prober generates evictions on both entry indices, we observe four peaks in total, where two correspond to the forward flow and two correspond to the backward flow; other entries are evicted at a low probability ($< 1.3\,\%$) due to noise.

Conclusion. The hash value that indexes the microflow cache is generated from the network packet header. Thus, entries with identical hash values may lead to a collision.

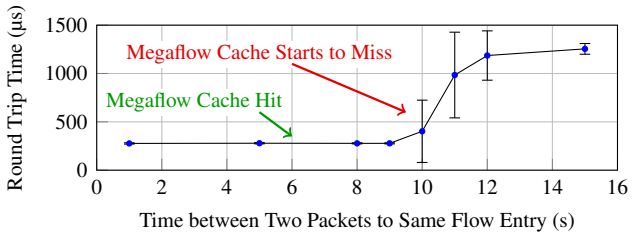


Figure 10: Megaflow timeout ($n = 100$ for each data point).

3.4 Megaflow cache

In this experiment, we characterize the megaflow cache. As it is accessed only upon microflow cache misses, all experiments below generate additional traffic to the packet handler to evict the microflow cache entry (based on colliding hash values), prior to megaflow cache measurements.

3.4.1 Megaflow Cache Timeout

Unlike the microflow cache, the megaflow cache periodically removes unused entries in subtables. As introduced in Section 2.3.2, the minimum timeout eviction interval is 10 seconds. In this experiment, we confirm this mechanism with a timeout eviction test on the megaflow cache. Figure 10 demonstrates RTT before and after timeout eviction, depending on the time between two consecutive packets. The prober first sends a packet to the packet handler on the server and starts waiting for a time interval (x-axis). Note that to eliminate microflow cache hits, the prober immediately evicts this packet flow from the microflow cache by sending a different but colliding packet. After the intended time interval has elapsed, the prober sends the second packet to the server and measures its RTT (y-axis) — a megaflow hit latency indicates the previous entry has not timed out, and a longer latency indicates the timeout has happened. For each time interval, we repeat the experiment 100 times. We observe that the megaflow cache starts to make timeout evictions after 10 s, as RTT increases at that point. When evictions start to happen, the standard deviation is high (as indicated by the error bars).

3.4.2 Subtable Reordering

As introduced in Section 2.3.2, the order of subtables depends on their hit counts. We characterize this correlation using the following approach. First, we control the background traffic that accesses 30 out of 40 subtables at fixed rates (from 1 to 10 packet / s on different subtables) to keep the same order among them. We follow the same approach in Section 3.4.1 to clear microflow cache entries. Then, the prober targets one of the remaining 10 subtables and alters the traffic intensity from 1 to 1000 packet / s. Figure 11 shows the RTT of the target subtable — the RTT reduces as the traffic intensity

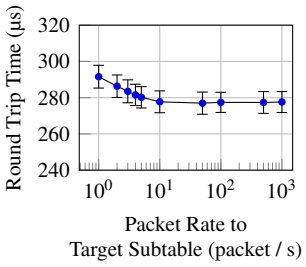


Figure 11: Megaflow cache reorder triggered by probing ($n = 100$ per data point).

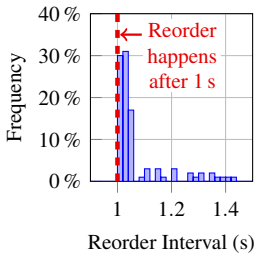


Figure 12: Subtable reorder intervals in megaflow cache ($n = 100$).

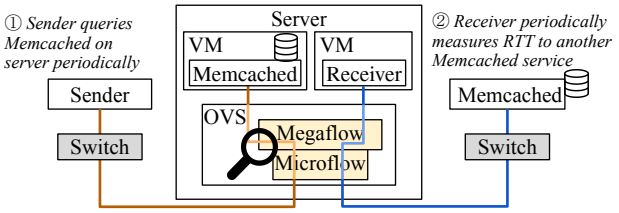


Figure 13: Setup of OVS covert channels.

increases. The RTT stays around 291 µs when the prober sends 1 packet / s and reduces to 277 µs as the prober hits the megaflow cache at a higher rate. This experiment confirms that the megaflow subtables are sorted by their hit counts.

We also investigate the time to reorder after the access count changes. Figure 12 shows the distribution of the subtable reorder interval. We find that the minimum reorder interval is 1 s and 86 % of the intervals are within 1.2 s. There is a tail that extends to 1.43 s, due to software variations. For example, OVS defers reordering if threads are busy with other tasks, such as OpenFlow rule insertion and slow path access.

Conclusion. The megaflow cache consists of a number of subtables. Subtables ordered to the front will be accessed first and have lower latencies. The megaflow cache periodically (every 1 – 1.43 s) sorts subtables by their hit counts to reduce the latency of frequently accessed subtables. Thus, the access latency of a subtable is correlated with its access frequency. Moreover, entries in the subtables get evicted after 10 s if they are not hit by future packets.

3.5 Summary of Attack Primitives in OVS

Based on the finding, we have identified three attack primitives in OVS.

1. **Timing differences among microflow cache, megaflow cache, and slow path** (Section 3.2): Hits and misses on microflow and megaflow caches cause latency differences in network round trip latency. Through the latency differences, an attacker can monitor the OVS cache status, further enabling them to infer network activities.
2. **Microflow cache hash collisions** (Section 3.3.2): The hash value that indexes the microflow cache is generated from five packet header fields. A packet with an identical header can lead to hash collisions in the microflow cache, which allows an attacker to infer the packet header fields.
3. **Megaflow cache subtable ordering** (Section 3.4.2): The megaflow cache periodically sorts subtables to minimize the hit latency. An attacker can manipulate the subtable ordering by controlling the rate of packets that go through the target subtable. Further, the 10 s timeout interval allows attackers to reset the megaflow cache state. However, this also requires attacks based on the megaflow cache to complete before the timeout.

Discussion. As discussed in Section 3.1, this setup assumes a simple one-hop network. Real-world environments may introduce additional noise due to more complex, multi-hop networks. We expect future work to explore the OVS attack primitives under more complex network configurations to better understand the impact of complex topologies. This work uses the OVS version with the DPDK datapath, which has both the microflow and megaflow caches. In comparison, the kernel version of OVS only has the megaflow cache. If a kernel version OVS is deployed, Attack Primitive 1 will have two levels of latency differences, Attack Primitive 2 will not apply due to the absence of the microflow cache, and Attack Primitive 3 will remain the same as the kernel version still maintains the megaflow subtables.

4 Remote Covert Channel

In this section, we introduce two remote covert channels using the microflow cache and megaflow cache in OVS.

4.1 Attack Model

The sender and the receiver are connected through a one-hop network via a switch, as Figure 13 illustrates. However, they do not have any direct connection for communication. The receiver is co-located with a Memcached service on the same server, running in separate VMs and connected with the same OVS on the host server. The sender has access to the co-located Memcached. The receiver also has access to another remote Memcached service that is external to its server, which is also connected through a one-hop network. Note that we use Memcached services for demonstration. The sender and the receiver can also leverage other network services such as databases and websites. This attack assumes that the sender and the receiver know the microflow cache entry and megaflow subtables they will access, according to the prior knowledge of the network setup (IP, port, and protocol). Cloud systems typically deploy a large number of OpenFlow rules, leading to numerous subtables in the megaflow cache [33, 45]. Therefore, the sender and the receiver assume substantial timing differences among subtable locations.

4.2 Attack Design

We detail the design of covert channel using both collisions in the microflow cache and subtables in the megaflow cache.

4.2.1 Microflow Cache

The microflow-cache-based covert channel uses the common Prime+Probe approach [28], using Attack Primitives 1 and 2. It exploits the latency difference between a microflow cache hit and a megaflow cache hit (i.e., upon microflow cache miss), as shown in Figure 7. Periodically, the sender queries the Memcached which is co-located with the receiver if it sends bit "1" to the receiver (step ①), where the query takes two flow entries (forward and backward) in the microflow cache, as discussed in Section 3.3.2. If the sender sends bit "0", it stays idle. The receiver knows the microflow cache entries that the sender uses based on prior knowledge (as discussed in Section 4.1. Thus, the receiver periodically probes the same microflow cache entries (step ②). Packet flows from the sender will evict the receiver's microflow cache entries. Thus, if the receiver measures a high RTT, the sender has sent bit "1", and otherwise, the sender has sent bit "0".

4.2.2 Megaflow Cache

The megaflow-cache-based side channel is based on Attack Primitives 1 and 3. We take the same approach in Section 3.4.2 to manipulate subtable ordering. Figure 14 shows the latency distributions of two subtable locations, where we measure each 100 times. Subtable location 1 is accessed 10 times every second to move the target subtable to the front, with an average latency of 276.90 µs ($\sigma = 1.6\,\%$); subtable location 2 is accessed 1 time per second, being left at the end, with an average latency of 283.04 µs ($\sigma = 1.5\,\%$). The sender first queries the Memcached service co-located with the receiver to create an initial ordering among subtables. Then, the sender targets one of the subtables and changes its location by controlling the packet rate towards this subtable (step ①). Note that the sender clears microflow cache entries to enable hits on megaflow subtables. If the receiver measures a low latency from a megaflow hit on the sender-controlled subtable, the sender has sent bit "1"; otherwise, the bit is "0" (step ②).

4.3 Setup

We follow the system configuration described in Section 3.1. On both Memcached services, a known key-value pair is stored before launching the covert channel. Thus, the sender and the receiver can access their known key-value pair in their separate Memcached services using `GET` requests. Note that the two Memcached instances are completely independent. The purpose of the `GET` requests is to generate network traffic (for the sender) and perform timing (for the receiver). For the microflow-cache-based covert channel, we evaluate both the 8k-entry and the 16k-entry versions. The threshold that determines microflow cache miss/hit is 273.63 µs, according to the latency distribution in Figure 7. Because the covert channel only accesses a specific microflow cache entry, the threshold is the same for different microflow cache sizes. To establish a megaflow-cache-based covert channel, the receiver determines the latency thresholds using the profiling approach in Section 3.4. In this experiment, we use the same OVS rules as Section 3.1. Thus, the threshold that determines 2 subtable locations is 279.97 µs, following the distributions in Figure 14. In both covert channels, every time, the sender transmits 2 bits "10" for the header and 8 bits for the payload (randomly generated in evaluation). Latency measurements are performed 50 times to transmit 1 bit for high accuracy.

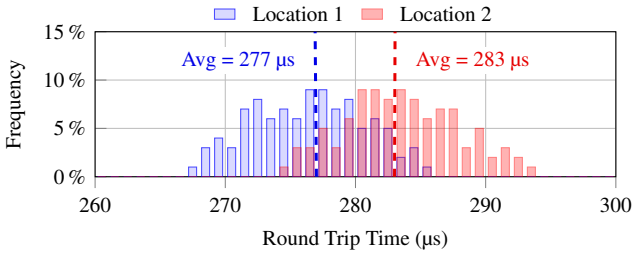


Figure 14: Megaflow cache hit latency ($n = 100$ per location).

Table 2: Covert channel bandwidth and accuracy.

Channel	Rep.	BW | σ (bit/s) | (%)	Accuracy (%) | σ (%) No Noise	Low	High
Microflow (8k-entry)	10	79.0 | 0.12	71.2 | 0.4	71.0 | 0.4	70.6 | 0.4
	30	26.3 | 0.08	86.9 | 0.3	85.6 | 0.4	85.0 | 0.4
	50	15.8 | 0.07	96.8 | 0.2	96.4 | 0.3	96.4 | 0.3
	75	11.9 | 0.05	98.1 | 0.2	96.6 | 0.2	96.5 | 0.3
	100	7.9 | 0.04	98.4 | 0.2	96.8 | 0.2	96.7 | 0.3
Microflow (16k-entry)	10	79.0 | 0.14	70.6 | 0.6	69.2 | 0.7	69.4 | 0.6
	30	26.3 | 0.11	86.5 | 0.4	85.1 | 0.5	85.6 | 0.5
	50	15.8 | 0.07	96.9 | 0.3	96.0 | 0.4	95.7 | 0.4
	75	11.9 | 0.05	98.0 | 0.2	96.7 | 0.3	96.4 | 0.3
	100	7.9 | 0.05	98.3 | 0.2	96.9 | 0.3	96.8 | 0.3
Megaflow	10	0.73 | 0.02	63.3 | 1.1	62.2 | 1.3	62.1 | 1.3
	30	0.72 | 0.01	75.3 | 0.7	71.5 | 0.9	71.9 | 0.9
	50	0.73 | 0.01	85.7 | 0.6	80.0 | 0.7	79.7 | 0.8
	75	0.73 | 0.01	87.0 | 0.6	85.6 | 0.6	85.3 | 0.6
	100	0.73 | 0.01	87.4 | 0.6	86.6 | 0.6	86.4 | 0.6

4.4 Results

In this section, we first present the bandwidth of covert channels and then conduct a sensitivity study.

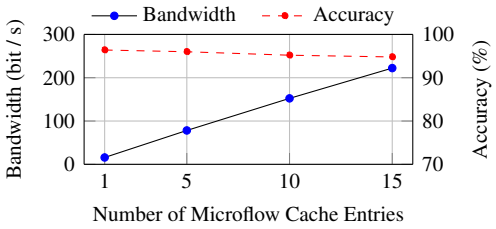


Figure 15: Covert channel using multiple microflow entries (8k-entry microflow cache, 50 repetitions, and high noise).

4.4.1 Covert Channel Bandwidth and Accuracy

Table 2 lists the results of the covert channels that use the microflow cache (both 8k and 16k versions) and the megaflow cache. All bandwidth measurements exclude header bits.

Microflow Cache. First, we evaluate the default microflow cache with 8k entries without noise from background traffic. With the default 50 times of measurement, the covert channel bandwidth is 15.8 bit / s. The accuracy is also high (96.8 %) as the latency of microflow cache hits and misses can be distinguished through multiple measurements, according to the characterization in Figure 7. Then, we increase the microflow cache to 16k entries. We find that the bandwidth and accuracy are close to the default 8k, as the microflow-cache-based covert channel only accesses a specific entry. We further perform a sensitivity test to analyze the impact of measurement repetitions (from 10 to 100 measurements). In both 8k and 16k-entry megaflow caches, an increased number of measurement repetitions improves the accuracy, but reduces the bandwidth as more measurements take longer.

Megaflow Cache. We next evaluate the megaflow cache without background noise. With the megaflow cache, the transmission rate is lower due to the long subtables reordering interval (Figure 12). With the default 50 measurement repetitions, the megaflow-based covert channel achieves 0.73 bit/s bandwidth and 85.7 % accuracy. We also perform a sensitivity test, showing that the accuracy also benefits from repeated measurements (87.4 % with 100 repetitions). However, unlike the microflow-based covert channel, its bandwidth remains unchanged because the megaflow cache reordering interval is over 1 s, allowing multiple measurements within this interval.

4.4.2 Sensitivity Studies

We finally conduct two sensitivity studies.

Noisy environment. We add background flows to evaluate our covert channels in a realistic environment. We use network traffic from the UNSW-NB15 dataset [35] as background noise and rank 60 s traces by their average packet rate. We choose P50 and P90 rates as the low-noise (40 packet / s) and high-noise (150 packet / s) scenarios, respectively. Table 2 also presents the accuracies of the covert channels under both noise levels. In all covert channels, accuracy drops are less than 2 %. For microflow-cache-based channels, the reason is the covert channel only targets a single entry. For the megaflow-cached-based channel, the microflow cache blocks most of the accesses, and thus, the noise has little impact.

Multiple microflow entries. To achieve a higher covert channel bandwidth, the sender and the receiver can use multiple microflow entries to transmit bits in parallel. We evaluated this approach using a configuration of 50 measurement repetitions under high-noise conditions with a standard 8k-entry microflow cache. Figure 15 shows that this approach achieves 222.3 bit / s bandwidth using 15 entries without major degradation of accuracy.

Table 3: Comparison with other remote covert channels.

Covert Channels	Bandwidth	Error
NetCAT [24]	16 kbit / s	0.2 %
Optane Persistent Memory [30]	10.01 bit / s	1.13 %
NetSpectre [47]	1.07 bit / s	<0.1 %
Memory Deduplication [48]	0.08 bit / s	0.6 %
This work: Microflow-cache-based (using 1 microflow entry)	15.8 bit / s	3.2 %

4.5 Discussion

In summary, we demonstrate covert channels using both the microflow and megaflow caches. All these covert channels are stealthy as they are remote.

Comparison with prior works. Table 3 compares our best-performing covert channel (microflow-cache-based) with several existing remote covert channels that leverage different components in computer systems to transmit data stealthily. For example, Kurth et al. [24] leverage Intel's data direct I/O technology (DDIO) that allows a secret channel between two remote clients, Schwarz et al. [47] implement a remote covert channel exploiting speculative execution in the server, Schwarzl et al. [48] exploit the timing difference due to copy-on-write page faults to build a remote covert channel, and Liu et al. [30] build a remote covert channel leveraging contentions in Optane persistent memory. Our covert channel bandwidths and error rates are comparable with these studies. The microflow-based covert channel achieves 1.58× to 197× higher bandwidth than the channels demonstrated in prior works [30, 47, 48]. Although NetCAT [24] achieves higher bandwidth, their approach transmits 64 bits in parallel. As shown in Figure 15, it is also possible to increase the bandwidth of the microflow-based covert channel by using multiple microflow cache entries in parallel.

Limitations. In our setup, OVS uses the DPDK datapath. When OVS uses the kernel datapath, there is only a megaflow cache. Consequently, the covert channel can only leverage the megaflow cache, which results in lower bandwidth.

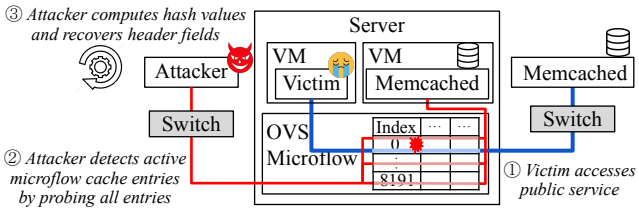


Figure 16: Setup for remote header recovery attack.

5 Remote Packet Header Recovery Attack

The hash value that indexes the microflow cache is generated from five packet header fields (Section 2.3.1). Therefore, it is possible to use microflow collisions to infer the header fields. In particular, with prior knowledge about the service that a victim uses, it is possible to acquire some of the fields and use the microflow cache collision to infer the remaining. We refer to this as a *packet header recovery attack*.

5.1 Attack Model

Figure 16 shows the attack model, where a remote user (i.e., the victim) and a Memcached service are located on the same server but isolated by separate VMs and connected with the OVS. The attacker has access to the co-located Memcached service via queries. On the other hand, the victim keeps accessing a publicly accessible service, which has the IP address, port, and protocol known to the attacker. Here, we also use Memcached queries as the victim's activity for demonstration. The attacker also knows the hash function used by the microflow cache, given that OVS is open-sourced.

5.2 Attack Design

The header recovery attack has two stages: a *probing stage* that tracks the victim's activities and an offline *hash computation* stage that infers victim's header fields based on probing.

Stage 1: Probing. This stage follows the Prime+Probe approach using attack primitives 1 and 2. The victim accesses a known remote service (step ①). A single packet flow looks up 2 microflow cache entries using two hash values, as discussed in Section 2.3.1. Thus, the attacker detects both entries accessed by the victim using the approach in Section 3.3.2: the attacker first primes all microflow cache entries and probes the evicted entries (step ②). To overcome noise, the attacker repeats this stage multiple times and selects the 2 entries with the highest average probing latencies. Then, for these 2 entries, the attacker generates probing packets with the same indices but hash values larger than existing probing packets to detect the eviction of their other associated entries, again selecting the 2 entries with the highest latency after repeated measurements. As the victim generates one forward request packet and receives one backward response packet, the attacker finally detects a total of 4 entries evicted by the victim.

Table 4: Time and accuracy of header recovery ($n = 100$).

Microflow	Rep.	Time (s) | σ (%)	No Noise	Low	High
8k-entry	10	83.4 | 0.21 %	36 %	33 %	31 %
	30	250.8 | 0.14 %	69 %	61 %	61 %
	50	417.6 | 0.14 %	91 %	82 %	80 %
	100	837.0 | 0.12 %	93 %	87 %	87 %
16k-entry	10	166.2 | 0.18 %	32 %	32 %	32 %
	30	501.0 | 0.14 %	70 %	59 %	60 %
	50	834.6 | 0.13 %	91 %	81 %	80 %
	100	1673.4 | 0.09 %	94 %	86 %	85 %

Stage 2: Hash computation. With the hash values that correspond to the evicted indices known, the second stage is to figure out the source IP address and port number in the packet header that generates these hash values. We first demonstrate this stage using the default 8k-entry microflow cache. Given the microflow hashing scheme (introduced in Section 2.3.1), the 32-bit source IP address and the 16-bit source port number form a total of 2^{48} possible pairs. Each detected microflow cache entry reduces the number of possible pairs by $2^{13}\times$, as a 13-bit hash value decides the microflow cache entry. In stage 1, the attacker detected 4 microflow cache entries. Thus, one possible pair of the victim's source IP address and port number remains. For different microflow cache sizes, as long as the size is at least 2^{12} entries (i.e., 4k), this approach can recover an exact packet header. Based on this idea, the attacker computes the hash values of all possible pairs of the victim's IP address and port number using multiple cores in parallel (step ③), and finds a common pair with a hash value that matches all 4 detected entries. This stage happens *offline*, after the attacker has completed stage 1.

5.3 Setup

This attack uses the same one-hop network system in Section 4.3. The victim keeps sending `GET` requests to a remote, publicly accessible Memcached service, where the destination IP address, the destination port number, and the protocol are known and remain the same throughout the attack. The latency threshold for microflow cache hit/miss follows Section 4.3. To evaluate the effectiveness of repetition, we test stage 1 using 10, 30, 50, and 100 repetitions.

5.4 Results

Table 4 presents the results. We first evaluate each configuration 100 times without background noise. With the default 8k-entry microflow cache, attack time grows with the number of measurement repetitions in stage 1, from 83.4 s (10 repetitions) to 837.0 s (100). The increase is due to stage 1,

as stage 2 is independent of repetitions (which takes 1.47 s, $\sigma = 1.32\,\%$). Under 8k-entry microflow cache, accuracy improves with repetitions, from 36 % at 10 to 93 % at 100. The 16k-entry cache doubles stage 1 time (twice as many entries to probe) but follows the accuracy trend, reaching 94 % at 100 repetitions. As this attack targets specific microflow entries, cache size has negligible impact on accuracy. We conclude that this attack can achieve a high accuracy with 50 or more repetitions in stage 1.

We next evaluate the attack under background noise, following the methodology in Section 4.4. Instead of using a fixed 60 s noise trace, we match noise trace length with the experiment duration but still use P50 and P90 packet rates for low- and high-noise settings. Because packet header recovery probes the whole microflow cache, high background noise leads to an accuracy drop of 9 %, but still achieving 85 %.

5.5 Discussion

Potential attacks leveraging packet header. An attacker can further incorporate existing activity probing attacks [11, 26, 37, 50, 53] or probe commonly used services to obtain the destination IP address, destination port number, and protocol since they can detect the website that the victim accesses. Then, the header recovery attack recovers the victim's IP address and port number. The recovered information can be exploited by the attacker to perform other attacks, such as scanning open ports on the user's machine to attack the user's insecure services [10, 51].

Comparison with prior works. This attack requires knowledge about the hash function that the microflow cache uses. Identifying the hash functions is a common approach before carrying out attacks. For example, Kayaalp et al. [21] and Gras et al. [15] reverse-engineer hash functions in the CPU cache to let the attacker manipulate hash collisions and infer the victim's information. OVS uses its open-sourced hash function by default but can be configured to use other hash functions [61]. Therefore, prior hash reverse-engineering approaches can be applied. Our header recovery attack computes all possible pairs of header fields. This approach is similar to the brute-force and dictionary attacks taken by prior works [5, 20, 39, 46]. In our attack, hash computation is fast and happens offline.

Limitations. The attacker repeats latency measurements to achieve a reliable accuracy, which results in a long probing time. Thus, the packet header recovery attack targets long-lived flows from the victim. Examples include video streaming, downloading, GenAI services such as coding assistance, online meeting, online gaming, and cloud-hosted workstations. Moreover, if OVS employs the kernel datapath instead of the DPDK datapath, the absence of the microflow cache disables this attack.

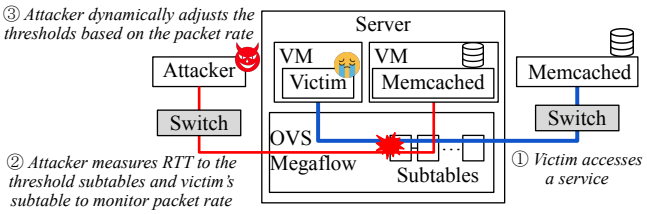


Figure 17: Setup for packet rate monitoring attack.

6 Remote Packet Rate Monitoring Attack

The megaflow cache in OVS periodically reorders subtables based on packet rate, where subtables with more accesses are ordered at the front to reduce access latency as discussed in Section 2.3.2. Therefore, the ordering enables the attacker to infer the approximate rate of victim flow's packet. We refer to this attack as a *remote packet rate monitoring attack*.

6.1 Attack Model

Like the packet header recovery attack, this attack targets a remote user, i.e., the victim, that is co-located with a Memcached service, both running in separate VMs and connected via OVS, as shown in Figure 17. The attacker has access to this co-located Memcached using queries. The victim accesses a service outside the server, whose IP address, port, and protocol are known to the attacker, e.g., using the packet header recovery attack in Section 5. Thus, the attacker is capable of inducing cache collisions in the microflow cache to access the same megaflow subtable utilized by the victim. We also use Memcached queries to illustrate the victim's activity and the attacker's probing access.

6.2 Attack Design

This attack is based on subtable reordering and the associated timing differences, leveraging Attack Primitives 1 and 3. It is enabled by two mechanisms, packet rate monitoring and adaptive monitoring of varying packet rates.

6.2.1 Rate monitoring

First, the attacker identifies megaflow subtables based on their latency differences using the same approach in Section 3.4.2 and 4.1. While the victim is accessing a service (step ①), the attacker periodically sends packets to 4 different subtables, where each of them receives a different packet rate. This way, the 4 subtables are ordered and form 4 thresholds (i.e., *Th 1* – *Th 4* in Figure 18). *Th 1* is the subtable with the highest rate of 4 packet / s and *Th 4* has the lowest rate of 1 packet / s. The attacker additionally sends the same set of rates to 5 *padding subtables* in between the 4 threshold subtables to preserve distinguishable latencies between the threshold subtables. To ensure the victim's packets end up accessing the megaflow

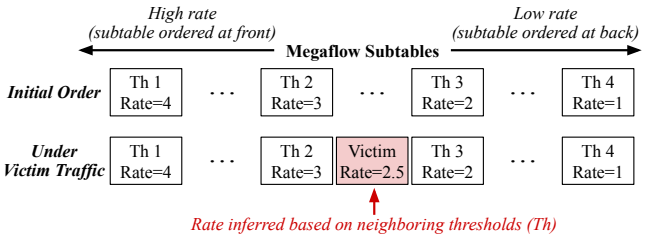


Figure 18: Subtable ordering for packet rate monitoring.

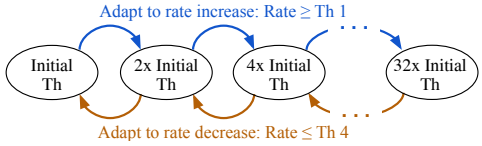


Figure 19: Dynamic threshold adjustment.

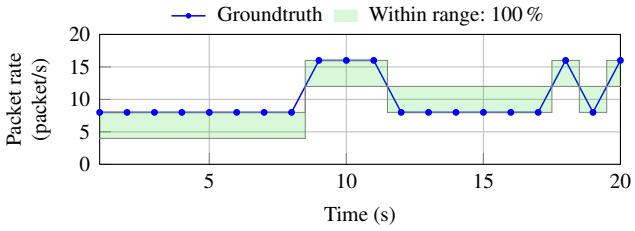
cache, the attacker sends a packet every 500 µs to evict the victim's flow in the microflow cache, which is sufficient to support the maximum monitoring rate in Section 6.2.2. This way, when the megaflow subtables get reordered, the subtable accessed by the victim will be ordered together with the attacker-controlled threshold subtables, as demonstrated in Figure 18. The attacker can probe the RTT of both the victim's subtable and threshold subtables and use the timing difference to infer the range of the victim's packet rate (step ②). The attacker's measurement repeats 100 times every second to achieve high accuracy. Because probing packets also access subtables, the attacker sends extra packets at the same rate to the padding subtables to maintain the relative order.

6.2.2 Adaptive threshold adjustment

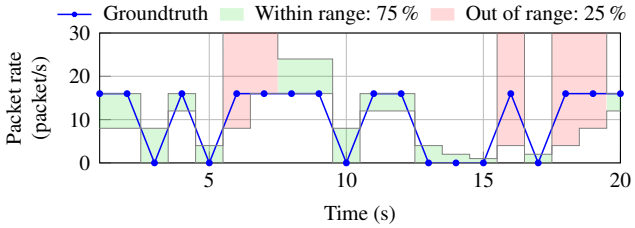
When the victim's flow has a higher packet rate than the rate of the highest threshold, the attacker can no longer track its range. To enable tracking of varying rates, the attacker adjusts the thresholds by changing the packet rate sent to the associated megaflow subtables. When the victim's detected packet rate goes beyond the highest threshold (*Th 1* in Figure 18), the adaptive threshold adjustment mechanism doubles the thresholds' packet rate. The maximum thresholds in this experiment are 32× of the initial values: 32, 64, 96, and 128 packet / s. Likewise, when the monitored rate drops below the lowest threshold (*Th 4*), the thresholds' packet rates are halved, until reaching the initial thresholds. The adjustment happens at the same rate as megaflow subtable reordering.

6.3 Setup

We follow the setup in Figure 17. The victim sends packets to a publicly accessible Memcached service, using `GET` requests. The attacker also measures RTT to the Memcached service in the server to recover the victim's packet rate. The OVS uses



(a) Flat packet rate.



(b) Varying packet rate.

Figure 20: Demonstrations of packet rate monitoring.

the same set of rules as Section 3.1. To reduce the noise of RTT, the attacker repeats all RTT measurements 100 times.

Dataset. We use the UNSW-NB15 dataset [35,36] to evaluate the monitoring accuracy. As packet rate monitoring targets relatively long-lasting flows, we filter out short (last less than 60 s) and inactive flows (< 1 packet / s on average). In total, the dataset has 30 flows for the experiment. These flows have packet rates up to 126 packet / s and average rates between 1 and 38.85 packet / s. We evaluate each flow for a duration of 60 s that captures its peak packet rate. The victim generates packets at the rate specified by the dataset.

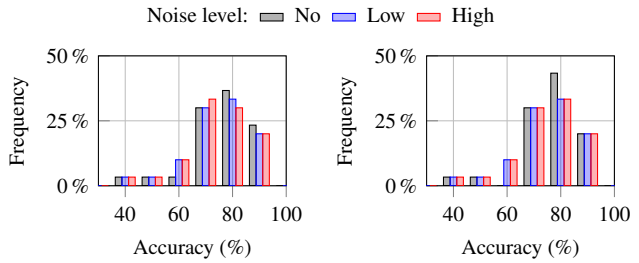
Accuracy metric. We classify an instance as successful when the groundtruth rate falls within the same threshold as the monitored rate. Conversely, misclassification or rates falling outside the minimum and maximum thresholds are classified as unsuccessful. We define monitoring accuracy as the ratio of successful monitoring instances over the total number of instances.

6.4 Results

Figures 20a and 20b show two 20 s traces. The blue line is the victim's ground truth; colored regions are the attacker's monitored ranges (green for successful and red for unsuccessful). Regions exceeding the y-axis indicate the rate exceeds the threshold. The attack is highly accurate on flat-rate segments as the ground truth stays within the threshold ranges (Figure 20a). When rates fluctuate, the success rate drops. For example, seconds 6–8 in Figure 20b shows a case where the victim's rate suddenly increases and the attacker's thresholds take 2 seconds to catch up.

Table 5: Effectiveness of defense mechanisms, i.e., attack accuracies after integrating the defense mechanisms.

Attacks	OVS Isolation	Microflow Randomization	Megaflow Randomization
Covert channel (Microflow)	49.60 % ($\sigma = 1.5$ %)	53.90 % ($\sigma = 2.6$ %)	—
Covert channel (Megaflow)	50.20 % ($\sigma = 1.4$ %)	51.80 % ($\sigma = 1.8$ %)	52.35 % ($\sigma = 2.7$ %)
Packet header recovery	0 %	5 %	—
Packet rate monitoring	10.83 % ($\sigma = 6.8$ %)	3.87 % ($\sigma = 1.4$ %)	13.33 % ($\sigma = 6.4$ %)



(a) 8k-entry microflow cache. (b) 16k-entry microflow cache.

Figure 21: Rate monitoring accuracy.

We first evaluate each flow 20 times and record its average accuracy in an environment without background noise. Figures 21a and 21b plot the accuracy distributions of the 30 flows for 8k and 16k-entry microflow caches. Averages are 71.92 % and 71.83 %, respectively, with minimal sensitivity to cache size as this attack relies on megaflow cache reordering. Spiky flows are harder to monitor, but 16.7% of flows have over 85 % accuracy. We then evaluate the accuracy under noise, following the same setup in Section 4.4. Figure 21 shows the accuracy of packet rate monitoring attack with the low (70.83 % and 70.53 %) and high noise (70.19 % and 69.81 %). The accuracy declines by 2 % as background traffic is mostly blocked by the microflow cache, without affecting packet rate monitoring that uses the megaflow cache.

6.5 Discussion

Potential attacks leveraging packet rates. The remote packet rate monitoring attack can further enable attacks such as traffic classification and activity profiling. The packet rate is a common feature in statistics-based network traffic classification and can be used to infer user's activities, as demonstrated by prior works [8, 12, 65].

Comparison with prior works. There are various activity profiling attacks in the literature [11, 25, 26, 37, 42, 50, 53, 62, 63]. For example, Naghibijouybari et al. [37] leverage GPU memory utilization and Tan et al. [53] exploit PCIe congestion due to the NIC to infer website activities. In comparison, our attack is highly stealthy as it is remote, without requiring the attacker to be physically co-located with the victim but only accessing a service that shares OVS with the victim. There is also a remote traffic monitoring attack [29], which exploits timing differences between SDN rule hits and misses. It tracks whether the victim's flow has occurred in a certain period of time to deduce the packet rate, which is a coarse-grained method. Our attack enables a real-time, more fine-grained packet rate recovery, as the attacker directly monitors the packet rate of the victim's network traffic.

Limitations. This attack achieves higher accuracy when the victim service has a stable packet rate. Therefore, the packet rate monitoring attack targets services that maintain a consistent packet rate over long periods of time, similar to the packet header recovery attack. In our experimental setting that uses OVS with the DPDK datapath, the microflow cache filters out most traffic and thus enabling the attacker to monitor the megaflow cache subtables with lower noise. OVS with a kernel datapath can perform a similar attack. However, because it does not have a microflow cache but only a megaflow cache, the attack is more susceptible to noise.

7 Attack Mitigations

In this section, we first describe three defense mechanisms that mitigate OVS side channels. Then, we discuss potential detection methods for side-channel attacks on OVS.

7.1 Defense Mechanisms

We implement three defense mechanisms and evaluate their effectiveness. Table 5 shows the effectiveness of these mechanisms against each of the three attacks. The evaluation follows the default configurations, where the OVS uses an 8k-entry microflow cache and the system has no background noise.

OVS Instance Isolation. The root cause of side channels in OVS is the sharing of caching structures. A direct approach is to eliminate this sharing by using separate instances of OVS concurrently on the server. We evaluate this approach by deploying two separate OVS instances for the victim and the co-located service that can be accessed by the attacker. Table 5 shows the effectiveness of isolation. Both covert channels have around 50 % accuracy. As each bit can be either a "0" or "1" with a 50 % chance due to the random payload in this experiment, 50 % is the accuracy of a random bit stream, indicating the effectiveness of this defense. The header recovery attack has a 0 % success rate, indicating that isolation eliminates this attack. The packet rate monitoring

attack has a 10.83 % accuracy. Because the isolation eliminates the victim's access to the attacker-monitored megaflow cache, almost all attacker's measurements report zero traffic (except for noise). As the evaluated dataset has 10.33 % zero traffic, this accuracy implies no actual victim traffic was successfully monitored. However, using individual OVS can lead to extra system overheads, such as memory and CPU cycles, and complicate network management and configuration.

Microflow Cache Hash Randomization. The microflow cache uses a simple hashing scheme, where hash collisions can be used to infer user activities and steal network header information. A possible approach to mitigate side channels is to randomize accesses. For each flow, instead of colliding into two fixed entries in the microflow cache, randomized algorithms can direct the flow to more potential locations, like those used in prior cache randomization approaches [27, 43, 58, 60]. This mitigation can increase the candidates from hash collisions, making it harder to guess the victim's flows. Table 5 shows the results when the randomization mechanism introduces 10× candidates. It effectively lowers the accuracy of both types of covert channels to almost 50 %, demonstrating effective mitigation. It significantly lowers the accuracy of the header recovery attacks — only 5 out of 100 trials by the attacker were successful. It also defends against the packet rate monitoring attack. Even though it uses the megaflow cache, randomization of the microflow cache disables precise microflow eviction.

Megaflow Cache Reordering Randomization. The megaflow cache is another structure that can be exploited by attackers, as its subtable reordering can still be used to infer or transmit information. One solution is to randomize the reordering interval. Instead of performing reordering every 1 s, the megaflow cache can randomly perform reordering at any time in a longer interval, reducing the correlation between access frequency and subtable location. Table 5 shows the results when the subtable reordering interval is randomly set between 1 and 10 s. Like microflow cache hash randomization, this defense lowers the megaflow-based covert channel accuracy to almost 50 %, indicating effective mitigation. It also defends against the packet rate monitoring attack. However, this defense is not effective against the microflow-based covert channel and the header recovery attack, as they do not rely on the megaflow cache.

7.2 OVS Attack Detection

An alternative approach is to detect side-channel attacks on OVS. These side-channel attacks mostly feature repeated accesses to a set of flows, such as performing Prime+Probe on certain flow entries in the microflow cache. Existing tools that support network traffic anomaly detection, such as Intrusion Detection/Prevention System (IDS/IPS) [7], Network Behavior Anomaly Detection (NBAD) [6, 40], and Security Information and Event Management (SIEM) [19], can identify suspicious network traffic introduced by the attacks. Moreover, prior research works also suggest detection methods focused on analyzing packet patterns [13, 22, 31]. Upon detection of malicious traffic, the host may drop packets from the sender or delay these packets. The host can also isolate packets from the potentially malicious users by directing them to a separate OVS instance.

8 Conclusions

The Open vSwitch (OVS) is a widely used software-based virtual switch. In this work, we investigate OVS from a security perspective. We first identify three attack primitives in the caching mechanism of OVS. Then, using these attack primitives, we demonstrate remote covert channels, a remote header recovery attack, and a remote packet rate monitoring attack on OVS. Our attacks leak user information remotely, without having the attacker co-located with the victim. Our study demonstrates that side channels via OVS can break the isolation in a virtualization environment.

Acknowledgement

We thank the anonymous reviewers and the shepherd for their valuable feedback, and Zixuan Wang and Korakit Seemakhupt for proofreading. This work was supported in part by funding from the Innovation for Defence Excellence and Security (IDEaS) program from the Department of National Defence (DND). This work was also supported by a Discovery Grant from the Natural Sciences and Engineering Research Council of Canada (NSERC).

Ethical Considerations

Decision to Conduct the Research. We investigated Open vSwitch (OVS) due to the increasing demands of virtualized environments, where OVS serves as one of the most widely deployed virtual switches. It has been adopted across numerous platforms, including XenServer 6.0, Xen Cloud Platform, OpenStack, openQRM, OpenNebula, and oVirt. Our study focuses on identifying potential vulnerabilities to bring this issue to the OVS developers. While vulnerability disclosure may expose potential attack vectors, we also present effective mitigation strategies and believe it is essential for stakeholders to understand both the risks and available defenses. Because OVS is extensively used in multi-tenant cloud environments, any side-channel risks that arise from resource sharing may introduce meaningful and severe security consequences.

Stakeholders. The identified attacks and vulnerabilities introduce significant security risks for cloud service providers that deploy OVS, as well as for the users who rely on these environments.

Impacts. As cloud environments continue to expand, virtual switches like OVS are widely deployed to improve network performance, and our findings highlight the security risks that accompany such deployments. First, users whose traffic shares the same OVS instance as an attacker may unintentionally leak sensitive information, such as IP addresses and port numbers, as shown in our study. Second, attackers can infer users' packet rates, potentially revealing further details about their activities. Finally, covert channels that we identify could enable unauthorized remote communication between distinct parties. We present the potential security risks and, at the same time, propose mitigation strategies that stakeholders can adopt to prevent the proposed attacks. By characterizing the attack primitives and providing defense mechanisms, we alert users to the associated risks.

Mitigation. As our proposed attacks pose severe threats to current network systems, we responsibly consider ways to mitigate these attacks. To address the identified vulnerabilities, we propose three defense mechanisms: isolating OVS instances, randomizing the microflow cache hash, and randomizing the interval of megaflow cache reordering. These mechanisms have been shared with the OVS developers, and we have evaluated their effectiveness. Isolation of OVS instances and randomization of the microflow cache hash successfully defend against all three attacks we propose. Randomization of the megaflow cache reordering interval mitigates megaflow-based attacks, preventing both the megaflow cache covert channel and packet-rate monitoring attacks. The corresponding results are reported in Section 7. Our defense mechanisms effectively mitigate the risks associated with sharing OVS instances on a server and prevent the attacks presented in this paper. Mitigation strategies not only prevent attacks but also protect sensitive user data, ensuring that the research contributes positively to the security of cloud environments.

Attack Setup. Our evaluation was conducted entirely within an isolated network environment, as described in Section 3. To avoid any possibility of interfering with external systems or networks, all cables connecting to outside networks were physically disconnected throughout the experiments, including during the characterization phase. The servers used in the evaluation were directly interconnected, and the physical switch enabling the 1-hop network was also fully isolated, with no external connections.

Open Science

In accordance with USENIX Security's Open Science policy, the artifacts for this work are publicly available at https://doi.org/10.5281/zenodo.17965902. The OpenFlow rule sets (ClassBench-ng) and the UNSW-NB15 dataset are available at https://classbench-ng.github.io/ and https://research.unsw.edu.au/projects/unsw-nb15-dataset, respectively.

References

[1] CVE-2017-9263. https://nvd.nist.gov/vuln/detail/CVE-2017-9263, 2017.

[2] CVE-2019-25076. https://nvd.nist.gov/vuln/detail/CVE-2019-25076, 2019.

[3] CVE-2020-35498. https://nvd.nist.gov/vuln/detail/CVE-2020-35498, 2020.

[4] CVE-2023-3966. https://nvd.nist.gov/vuln/detail/CVE-2023-3966, 2023.

[5] Leon Bošnjak, J Sreš, and Bosnjak Brumen. Brute-force and dictionary attack on hashed real-world passwords. In *41st international convention on information and communication technology, electronics and microelectronics (MIPRO)*, pages 1161–1166, 2018.

[6] Cisco. Cisco Secure Network Analytics. https://www.cisco.com/site/us/en/products/security/security-analytics/index.html, 2025.

[7] Cisco. Snort. https://www.snort.org/, 2025.

[8] Manuel Crotti, Maurizio Dusi, Francesco Gringoli, and Luca Salgarelli. Traffic classification through simple statistical fingerprinting. *ACM SIGCOMM Computer Communication Review*, 37(1):5–16, 2007.

[9] Levente Csikor, Vipul Ujawane, and Dinil Mon Divakaran. On the feasibility and enhancement of the tuple space explosion attack against Open vSwitch. *CoRR*, abs/2011.09107, 2020.

[10] Marco De Vivo, Eddy Carrasco, Germinal Isern, and Gabriela O De Vivo. A review of port scanning techniques. *ACM SIGCOMM Computer Communication Review*, 29(2):41–48, 1999.

[11] Debopriya Roy Dipta and Berk Gulmezoglu. DF-SCA: Dynamic frequency side channel attacks are practical. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 841–853, 2022.

[12] Jeffrey Erman, Anirban Mahanti, Martin Arlitt, Ira Cohen, and Carey Williamson. Semi-supervised network traffic classification. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 369–370, 2007.

[13] Minghui Gao, Li Ma, Heng Liu, Zhijun Zhang, Zhiyan Ning, and Jian Xu. Malicious network traffic detection based on deep neural networks and association analysis. *Sensors*, 20(5):1452, 2020.

[14] Nakul Garg, Irtaza Shahid, Erin Avllazagaj, Jennie Hill, Jun Han, and Nirupam Roy. ThermWare: Toward side-channel defense for tiny IoT devices. In *Proceedings of the 24th International Workshop on Mobile Computing Systems and Applications (HotMobile)*, page 81–88, 2023.

[15] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 955–972, 2018.

[16] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on AES to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505. IEEE, 2011.

[17] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. Adversarial prefetch: New cross-core cache side channel attacks. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1458–1473. IEEE, 2022.

[18] Mathew Hogan, Yan Michalevsky, and Saba Eskandarian. DBREACH: Stealing from databases using compression side channels. In *IEEE Symposium on Security and Privacy (S&P)*, pages 182–198, 2023.

[19] IBM. IBM QRadar SIEM. https://www.ibm.com/products/qradar-siem, 2025.

[20] Mobin Javed and Vern Paxson. Detecting stealthy, distributed SSH brute-forcing. In *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security (CCS)*, page 85–96, 2013.

[21] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[22] Myung-Sup Kim, Hun-Jeong Kong, Seong-Cheol Hong, Seung-Hwa Chung, and James W Hong. A flow-based method for abnormal network traffic detection. In *IEEE/IFIP network operations and management symposium (IEEE Cat. No. 04CH37507)*, volume 1, pages 599–612, 2004.

[23] Taehun Kim and Youngjoo Shin. ThermalBleed: A practical thermal side-channel attack. *IEEE Access*, 10:25718–25731, 2022.

[24] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical cache attacks from the network. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 20–38, 2020.

[25] Alexander S La Cour, Khurram K Afridi, and G Edward Suh. Wireless charging power side-channel attacks. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 651–665, 2021.

[26] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. Stealing webpages rendered on your browser by exploiting GPU vulnerabilities. In *IEEE Symposium on Security and Privacy (S&P)*, pages 19–33. IEEE, 2014.

[27] Fangfei Liu and Ruby B Lee. Random fill cache architecture. In *47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 203–215, 2014.

[28] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *IEEE symposium on security and privacy (S&P)*, pages 605–622, 2015.

[29] Sheng Liu, Michael K Reiter, and Vyas Sekar. Flow reconnaissance via timing attacks on sdn switches. In *2017 IEEE 37th international conference on distributed computing systems (ICDCS)*, pages 196–206. IEEE, 2017.

[30] Sihang Liu, Suraaj Kanniwadi, Martin Schwarzl, Andreas Kogler, Daniel Gruss, and Samira Khan. Side-channel attacks on optane persistent memory. In *32nd USENIX Security Symposium (USENIX Security)*, pages 6807–6824, 2023.

[31] Nikita Lyamin, Denis Kleyko, Quentin Delooz, and Alexey Vinel. AI-based malicious network traffic detection in VANETs. *IEEE Network*, 32(6):15–21, 2018.

[32] Jiří Matoušek, Gianni Antichi, Adam Lučanský, Andrew W. Moore, and Jan Kořenek. ClassBench-ng: Recasting ClassBench after a decade of network evolution. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 204–216, 2017.

[33] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM computer communication review*, 38(2):69–74, 2008.

[34] Sebastiano Miano, Fulvio Risso, Mauricio Vásquez Bernal, Matteo Bertrone, and Yunsong Lu. A framework for eBPF-based network functions in an era of microservices. *IEEE Transactions on Network and Service Management*, 18(1):133–151, 2021.

[35] Nour Moustafa and Jill Slay. UNSW-NB15: A comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set). In *2015 military*

communications and information systems conference (MilCIS), pages 1–6. IEEE, 2015.

[36] Nour Moustafa and Jill Slay. The evaluation of network anomaly detection systems: Statistical analysis of the UNSW-NB15 data set and the comparison with the KDD99 data set. *Information Security Journal: A Global Perspective*, 25(1-3):18–31, 2016.

[37] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: GPU side channel attacks are practical. In *Proceedings of the ACM SIGSAC conference on computer and communications security (CCS)*, pages 2139–2153, 2018.

[38] Open vSwitch. Open vSwitch with DPDK. https://docs.openvswitch.org/en/latest/intro/install/dpdk/, 2023.

[39] Jim Owens and Jeanna Neefe Matthews. A study of passwords and methods used in brute-force ssh attacks. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2008.

[40] Palo Alto Networks. Cortex XDR. https://www.paloaltonetworks.com/cortex/cortex-xdr, 2025.

[41] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The design and implementation of open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 117–130, 2015.

[42] Yi Qin and Chuan Yue. Website fingerprinting by power estimation based side-channel attacks on Android 7. In *17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 1030–1039, 2018.

[43] Moinuddin K Qureshi. CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 775–787, 2018.

[44] Mark Randolph and William Diehl. Power side-channel attack analysis: A review of 20 years of study for the layman. *Cryptography*, 4(2):15, 2020.

[45] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. Scaling Open vSwitch with a computational cache. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1359–1374, 2022.

[46] Ragil Saputra, Beta Noranita, et al. Analysis of GPGPU-based brute-force and dictionary attack on SHA-1 password hash. In *3rd International Conference on Informatics and Computational Sciences (ICICoS)*, pages 1–4. IEEE, 2019.

[47] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read arbitrary memory over network. In *Computer Security–ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part I 24*, pages 279–299, 2019.

[48] Martin Schwarzl, Erik Kraft, Moritz Lipp, and Daniel Gruss. Remote memory-deduplication attacks. In *29th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2022.

[49] Aria Shahverdi, Mahammad Shirinov, and Dana Dachman-Soled. Database reconstruction from noisy volumes: A cache side-channel attack on SQLite. In *30th USENIX Security Symposium (USENIX Security)*, pages 1019–1035, 2021.

[50] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *28th USENIX Security Symposium (USENIX Security)*, pages 639–656, 2019.

[51] Arunan Sivanathan, Hassan Habibi Gharakheili, and Vijay Sivaraman. Can we classify an IoT device using TCP port scan? In *IEEE International Conference on Information and Automation for Sustainability (ICIAfS)*, pages 1–4, 2018.

[52] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging P4 programs with vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, page 518–532, 2018.

[53] Mingtian Tan, Junpeng Wan, Zhe Zhou, and Zhou Li. Invisible probe: Timing attacks with PCIe congestion side-channel. In *IEEE Symposium on Security and Privacy (S&P)*, pages 322–338. IEEE, 2021.

[54] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. Revisiting the open vSwitch dataplane ten years later. In *Proceedings of the ACM SIGCOMM Conference*, page 245–257, 2021.

[55] Vašek Šraier. An Open vSwitch security feature causes a security problem. Here's how to prevent it, 2024.

[56] Ying Wan, Haoyu Song, Yang Xu, Yilun Wang, Tian Pan, Chuwen Zhang, and Bin Liu. T-cache: Dependency-free ternary rule cache for policy-based forwarding. In

IEEE Conference on Computer Communications (INFOCOM), pages 536–545, 2020.

[57] Ying Wan, Haoyu Song, Yang Xu, Chuwen Zhang, Yi Wang, and Bin Liu. Adaptive batch update in TCAM: How collective optimization beats individual ones. In *IEEE Conference on Computer Communications (INFOCOM)*, pages 1–10, 2021.

[58] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, pages 494–505, 2007.

[59] Lingxiao Wei, Bo Luo, Yu Li, Yannan Liu, and Qiang Xu. I know what you see: Power side-channel attack on convolutional neural network accelerators. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 393–406, 2018.

[60] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting cache attacks via cache set randomization. In *28th USENIX Security Symposium (USENIX Security)*, pages 675–692, 2019.

[61] Yunhong Xu, Keqiang He, Rui Wang, Minlan Yu, Nick Duffield, Hassan Wassel, Shidong Zhang, Leon Poutievski, Junlan Zhou, and Amin Vahdat. Hashing design in modern networks: Challenges and mitigation techniques. In *USENIX Annual Technical Conference (ATC)*, pages 805–818, 2022.

[62] Qing Yang, Paolo Gasti, Kiran Balagani, Yantao Li, and Gang Zhou. USB side-channel attack on Tor. *Computer Networks*, 141:57–66, 2018.

[63] Qing Yang, Paolo Gasti, Gang Zhou, Aydin Farajidavar, and Kiran S Balagani. On inferring browsing activity on smartphones via USB power analysis side-channel. *IEEE Transactions on Information Forensics and Security*, 12(5):1056–1066, 2016.

[64] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security)*, pages 719–732, 2014.

[65] Jun Zhang, Xiao Chen, Yang Xiang, Wanlei Zhou, and Jie Wu. Robust network traffic classification. *IEEE/ACM transactions on networking*, 23(4):1257–1270, 2014.

[66] Mark Zhao and G Edward Suh. FPGA-based remote power side-channel attacks. In *IEEE Symposium on Security and Privacy (S&P)*, pages 229–244, 2018.